МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Рандомизированное бинарное дерево поиска (демонстрация)

Студент гр. 8304 _____ Бутко А.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2019

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент: Бутко А.М.

Группа: 8304

Тема работы: Рандомизированное бинарное дерево поиска (демонстрация)

Исходные данные:
Пользователю предоставляется выбор между вводом элементов из файла (каждый элемент считывается с новой строчки) и вводом элементов с консоли. Независимо от выбранного режима ввода, пользователю предоставляется возможность редактировать дерево (добавлять новые элементы, удалять элементы из РБДП, а так же вывод и выход из программы с выводом получившегося дерева).

Предполагаемый объем пояснительной записки:
Не менее 25 страниц.

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата:

Студент		Бутко А.М.
Преподаватель		Фирсов М.А.

АННОТАЦИЯ

Было изучено и реализовано рандомизированное бинарное дерево поиска (РБДП) на ЯП C++, реализованы основные функции работы с РБДП (добавление элемента, поиск элемента, удаление элемента, печать всего РБДП), меню для работы с ним с элементарным UX.

С точки зрения типа работы «Визуализация» было проведено полное логгирование выполняемых методов, их демонстрация, а так же вывод бинарного дерева (по уровням) в консоль.

Работа выполнялась в среде разработки CLion.

SUMMARY

Random Binary Search Tree (RBST) was explored and realised on C++ programming language, main functions for working with RBST was realised (adding of element, searching of element, element deleting, printing of all RBST), and was created menu with elementary UX for work with it.

From point of view on the type of course work «Visualisation» the full-logging of used methods was conducted, demonstration of it and outputting RBST by level in console.

Coursework was carried out in IDE CLion.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы было создание рандомизированного бинарного дерева поиска — структуры данных, реализующей бинарное дерево поиска.

Использование данной структуры данных производительней использования стандартного БДП, т.к время выполнения алгоритмов, работающих с БДП зависит от формы деревьев. РБДП позволяет избежать «худшего» варианта построения БДП. Как известно, можно подобрать такую последовательность операций с бинарным деревом поиска в наивной реализации, что его глубина будет пропорциональна количеству ключей, а следовательно операции будут выполняться за O(n). Поэтому, если поддерживать инвариант "случайности" в дереве, то можно добиться того, что математическое ожидание глубины дерева будет небольшим. Идея RBST состоит в том, что хранимое дерево постоянно является рандомизированным бинарным деревом поиска.

1. ОСНОВНЫЕ ОПЕРАЦИИ В РБДП

1.1. Хранение

В классе RBST был реализована структура, представляющая собой узлы дерева. Каждый узел РБДП будет содержать ключ element, указатели на сыновьи элементы left и right, а так же поле N, в котором будет храниться размер (в вершинах дерева) с корнем в данном узле. Код структуры представлен в листинге 1.1.1.

Листинг 1.1.1 — Структура для представления узлов РБДП

```
1.   struct Node
2.   {
3.       Elem element;
4.       std::shared_ptr<Node> left;
5.       std::shared_ptr<Node> right;
6.       int N;
7.       explicit Node (Elem value)
8.       {
9.           element = value,
10.          left = right = 0;
11.          N = 1;
12.      }
13.  };
```

Так же были реализованы функции getsize() и fixN() (код представлен в листинге 1.1.2), т.к. мы вынуждены хранить размер дерева, а так же корректировать размер самого дерева при его изменении в функциях работы с ним.

Листинг 1.1.2 — Функции для фиксации текущего размера дерева

```
1.   int getSize(link root)
2.   {
3.       if (!root) return 0;
4.       return root->N;
5.   }
6.
7.   void fixN(link root)
8.   {
9.       root->N = getSize(root->left) + getSize(root->right) + 1;
10.  }
```

1.2. Поиск

Поиск ключа в рандомизированном дереве поиска ничем не отличается от поиска ключа в стандартном дереве поиска.

Для поиска элемента в бинарном дереве поиска был реализован метод класса RBST search() (код метода (здесь и далее кол представлен без логгирования) представлен в листинге 1.2.1), который принимает в качестве параметров корень дерева и искомый ключ. Для каждого узла функция сравнивает значение его ключа с

искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева.

Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы *O(H)*, где *H* — высота дерева.

Листинг 1.2.1 — Метод поиска ключа в РБДП

```
1.   link search(link root, Elem value)
2.   {
3.      if (!root) return 0;
4.      if (value == root->element) return root;
5.      if (value < root->element) return search(root->left, value);
6.      else return search(root->right, value);
7.   }
```

1.3. Вставка

Рассмотрим рекурсивный алгоритм вставки ключа value в RBST, состоящее из N вершин. С вероятностью $\frac{1}{N+1}$ вставим ключ в корень дерева (разделим дерево по данному ключу и подвесим получившиеся деревья к новому корню), используя метод insertRoot() (код метода см. в листинге 1.3.1), который принимает в качестве параметров корень дерева и добавляемый ключ.

Листинг 1.3.1 — Метод вставки ключа в корень дерева

```
1.   link insertRoot(link root, Elem value)
2.   {
3.       if (!root)return std::unique_ptr<Node>(new Node(value));
4.       if (root->element > value)
5.       {
6.           root->left = insertRoot(root->left, value);
7.           return rotateRight(root);
8.       }
9.       else
10.      {
11.          root->right = insertRoot(root->right, value);
12.          return rotateLeft(root);
13.      }
14.  }
```

Если предположить, что ключ вставляемого элемента больше ключа в корне, то создание нового дерева можно было бы начать с помещения нового элемента в новый корневой узел, устанавливая старый корень в качестве левого поддерева, а правое поддерево старого корня в качестве правого поддерева. Однако, правое поддерево может содержать некоторые меньшие ключи, поэтому потребуется выполнить дополнительные действия (аналогично в случае, если ключ вставляемого элемента

меньше ключа в корне). Перемещения всех узлов с меньшими ключами в левое поддерево и всех узлов с большими ключами в правое в общем случае оказываются сложным преобразованием, т.к. узлы, которые нужно переместить, могут быть разбросаны по всему пути поиска для вставляемого узла.

Во избежание проблем, описанных выше, прибегнем к рекурсивному способу *ротации* дерева (rotation). Например, ротация вправо затрагивает корень и левый дочерний узел; ротация помещает корень справа, изменяя на обратное направление левой связи корня: перед ротацией она указывает от корня к левому дочернему узлу, а после ротации — от левого дочернего узла (нового корня) к старому корню(правый дочерний узел нового корня). Ротация — это локальное изменение, затрагивающее только три связи и два узла, которое позволяет перемещать узлы по деревьям, не изменяя глобальные свойства порядка. Исходный код функций ротации влево и вправо представлены в листинге 1.3.2.

Теперь, когда у нас есть функции ротации вправо(влево), можно описать полный алгоритм вставки ключа в корень. Сначала рекурсивно вставляем корень в левого (правого) поддерева (в зависимости от результатов сравнения нового ключа с корневым ключом), затем выполняем правый (левый) поворот при помощи ротации, который поднимает нужный нам ключ в корень дерева.

Листинг 1.3.2 — Функции ротации дерева

```
1.   link rotateRight(link root)
2.   {
3.       link tmp = root->left;
4.       if (!tmp) return root;
5.       root->left = tmp->right;
6.       tmp->right = root;
7.       tmp->N = root->N;
8.       fixN(root);
9.       return tmp;
10.  }
11.
12.  link rotateLeft(link root)
13.  {
14.      link tmp = root->right;
15.      if (!tmp) return root;
16.      root->right = tmp->left;
17.      tmp->left = root;
18.      tmp->N = root->N;
19.      fixN(root);
20.      return tmp;
21.  }
```

С вероятностью $1 - \frac{1}{N+1} = \frac{N}{N+1}$ вставим ключ в правое поддерево, если он больше корня, или в левое поддерево, если меньше. В листинге 1.3.3 представлен код метода insert(), который принимает в качестве параметров корень дерева и добавляемый ключ.

Листинг 1.3.3 — Метод вставки ключа в правое (левое) поддерево

```
1.   link insert(link root, Elem value)
2.   {
3.       if (!root)return std::unique_ptr<Node>(new Node(value));
4.       if (rand()%(root->N + 1) == 0)return insertRoot(root, value);
5.       if (root->element > value)root->left = insert(root->left, value);
6.       else root->right = insert(root->right, value);
7.       fixN(root);
8.       return root;
9.   }
```

1.4. Удаление

Рассмотрим алгоритм удаления элемента из РБДП. Алгоритм удаления использует операцию merge() (код представлен в листинге 1.4.1) — слияние двух деревьев, удовлетворяющих условию: все ключи в одном из деревьев меньше ключей во втором. В качестве корня нового поддерева можно взять любой из двух корней, например левый (если левого поддерева равен n, а правого — m, тогда первый корень выбирается новым корнем с вероятностью $\frac{n}{m+n}$, второй с вероятностью $\frac{m}{m+n}$ соответственно). Тогда левое поддерево останется как и есть, а справа будет подвешено объединение двух деревьев — правого поддерева левого поддерева и всего правого поддерева.

Листинг 1.4.1 — Функция слияния двух деревьев

```
1.   link merge(link left, link right)
2.   {
3.       if(!left) return right;
4.       if(!right) return left;
5.       if(rand()%(left->N + right->N) < left->N)
6.       {
7.           left->right = merge(left->right, right);
8.           fixN(left);
9.           return left;
10.      }
11.      else
12.      {
13.          right->left = merge(left, right->left);
14.          fixN(right);
15.          return right;
16.      }
```

```
17.  }
```

Для того, чтобы удалить некоторый ключ value из RBST сначала найдём вершину с этим ключом в дереве, используя стандартный алгоритм поиска. Затем объединяем левое и правое поддеревья найденного узла, удаляем узел и возвращаем корень объединенного дерева. Код метода remove(), принимающего в качестве параметров корень дерева и удаляемый ключ, представлен в листинге 1.4.2.

Листинг 1.4.2 — Метод поиска и удаления ключа

```
1.   link remove(link root, int value)
2.   {
3.       if(!root) return root;
4.       if(root->element == value)
5.       {
6.           link q = merge(root->left, root->right);
7.           return q;
8.       }
9.       else if(value < root->element)
10.          root->left = remove(root->left, value);
11.      else
12.          root->right = remove(root->right, value);
13.      return root;
14.  }
```

2. РАБОТА С ПРОГРАММОЙ

2.1. Интерфейс программы

При запуске программы перед пользователем появляется меню, с помощью которого он должен выбрать, откуда именно считывать рандомизированное бинарное дерево поиска. Данное меню представлено на рис. 2.1.1.

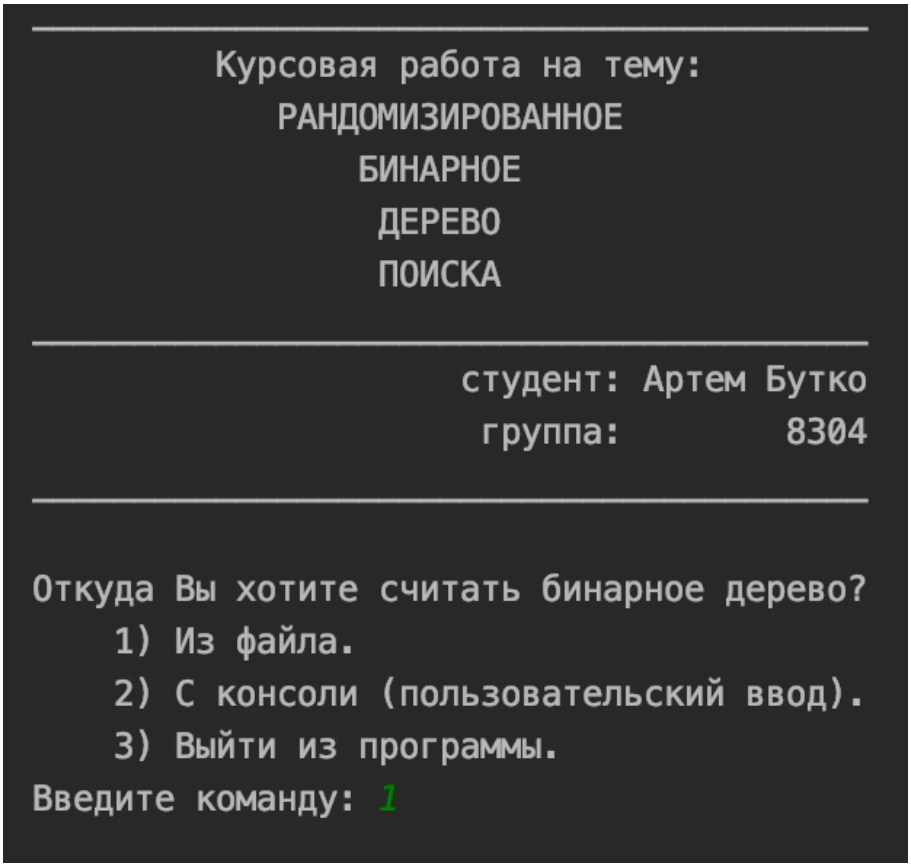

Рисунок 2.1.1 — Меню выбора для считывания РБДП

При наборе пользователем команды №1 будет запрошен путь до файла считывания, затем проводится логгирование считывания и выводится меню редактирования РБДП (см. рис. 2.1.2). При выборе 2 команды пользователю сразу предоставляется меню редактирования РБДП. При наборе пользователем 3 команды программа завершится. В противном случае, если будет выбрана некорректная команда, программа оповестит пользователя и запросит новую команду.

В меню редактирования РБДП пользователю предоставляется возможность работать с деревом напрямую (добавлять эл-ты, удалять эл-ты, печать РБДП). При выходе из редактирования программа выведет конечное РБДП и завершится. При наборе некорректной команды, программа запросит команду заново.

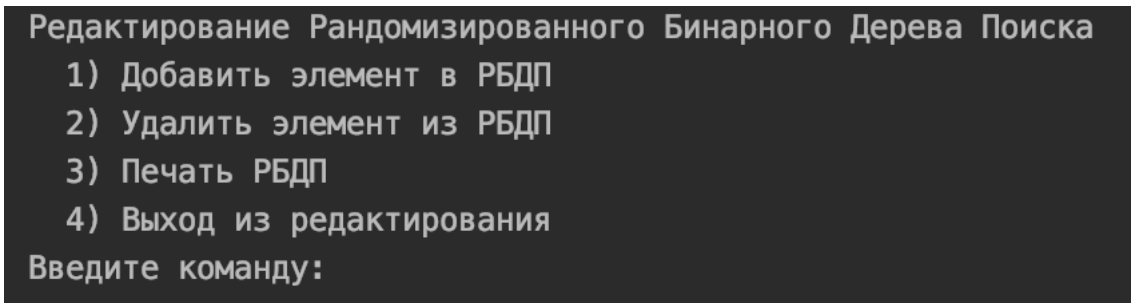

Рисунок 2.1.2 — Меню редактирования РБДП

2.2. Структура программы

В теле главной функции main файла main.cpp (исходный код представлен в приложении А) находится конструкция switch, из которой вызывается функция для чтения из файла readFromFile(), и функция readFromConsole() для считывания с консоли (меню редактирования РБДП).

Функция readFromFile() запрашивает у пользователя путь до файла, из которого построчно считываются и добавляются ключи РБДП с последующим логгированием. Затем печатается дерево и вызывается функция readFromConsole(), затем программа завершается.

Функция readFromConsole() запускает конструкцию switch, которая является меню редактирования РБДП. Команда №1 (добавление элемента) запрашивает ключ и запускает функцию класса RBST (исходный код представлен в приложении Б) searchAndInsertElement(), который в свою очередь ведет поиск ключа в дереве с помощью метода search() (см. стр. 6) и добавляет элемент в дерево с помощью метода insert() (см. стр. 7), если не был найден дубликат. Команда №2 (удаление элемента) запрашивает ключ и запускает функцию класса RBST deleteElement(), который в свою очередь вызывает метод remove() (см. стр. 9) . Команда №3 (печать РБДП) вызывает функцию класса RBST printTree(), которая обращается к методу print().

Метод print() (код представлен в листинге 2.2.1) совершает прямой обход дерева, печатая ключи узлов дерева. Метод принимает в качестве аргумента глубину нахождения узла и корень дерева. При выводе глубина хранения узла представлена количеством «-» перед элементом.

Листинг 1.4.2 — Метод поиска и удаления ключа

```
1.    void print(link root, int i)
2.    {
3.        if (root->right != 0) print(root->right, i+1);
4.        for (int j = 0 ; j < i; ++j)
5.            std::cout << " - ";
6.        std::cout << "(" << root->element << ")" << std::endl;
7.        if (root->left != 0) print(root->left, i+1);
8.    }
```

3. ТЕСТИРОВАНИЕ

3.1. Считывание из файла

Введем в меню выбора считывания считывание из файла, в котором построчно вписаны ключи в порядке: 1, 2, 3, 4, 1, 2. Результат считывания представлен на рисунке 1 приложения В (стр. рис. 1).

Теперь удаляем элемент 2 из считанного РБДП. Результат удаления представлен на рисунке 3.1.1.

```
Редактирование Рандомизированного Бинарного Дерева Поиска
  1) Добавить элемент в РБДП
  2) Удалить элемент из РБДП
  3) Печать РБДП
  4) Выход из редактирования
Введите команду: 2

Введите элемент: 2
 * удаляемый  ключ 2 больше, чем ключ текущего узла 1, спускаемся на правое поддерево. *
 * удаляемый ключ 2 найден в дереве, начинается процедура удаления. *

1 – Добавить, 2 – Удалить, 3 – Печать, 4 – Выход
Введите команду: 3

 – (4)
 –  – (3)
(1)
```

Рисунок 3.1.1. — Результат удаления узла.

Выведенная конструкция соответствует дереву на рис. 3.1.2.

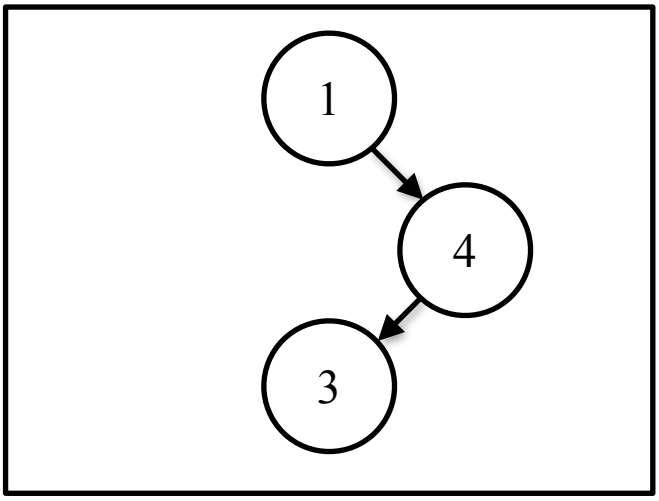

Рисунок 3.1.2 — Визуализация получившегося РБДП

3.2. Пользовательский ввод

Теперь попробуем ввести дерево самостоятельно. Входные данные: 1, 2, 3, 4, 5, 6, 7. Результат представлен на рисунке 3.2.1. Выведенная конструкция соответствует дереву на рис. 3.2.2.

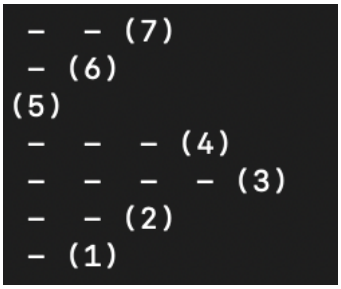

Рисунок 3.2.1. — Результат ввода данных

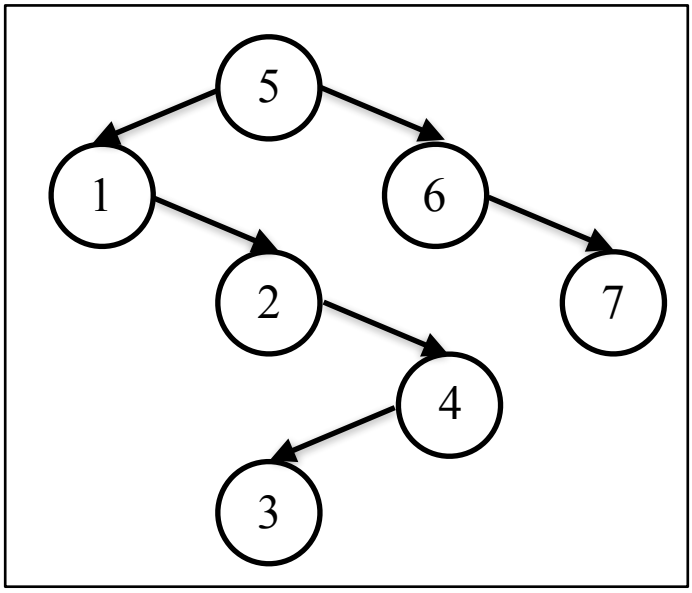

Рисунок 3.2.2. — Полученное дерево

ЗАКЛЮЧЕНИЕ

Было изучено и реализовано рандомизированное бинарное дерево поиска, а так же реализованы основные операции работы с РБДП (добавление, удаление эл-тов, печать дерева, поиск). Был показан процесс создания дерева с помощью логгирования.

РБДП — полезная структура данных, позволяющая избежать плохого построения стандартного бинарного дерева поиска, а так же выполняющая поиск, вставку и удаление с логарифмической скоростью.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Седжвик Р. Фундаментальные алгоритмы на C++. Части 1—4. К.: ДиаСофт, 2002. 688 с.

2. https://neerc.ifmo.ru/wiki/

3. https://habr.com/

4. https://ru.wikipedia.org/

ПРИЛОЖЕНИЕ А

ФАЙЛ MAIN.CPP

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.
5.  #include "RBST.h"
6.
7.  void readFromConsole(RandomBinarySearchTree<char> Tree)
8.  {
9.      char choice,
10.          element;
11.     std::cout << "Редактирование Рандомизированного Бинарного Дерева Поиска\n"
12.                  "  1) Добавить элемент в РБДП\n"
13.                  "  2) Удалить элемент из РБДП\n"
14.                  "  3) Печать РБДП\n"
15.                  "  4) Выход из редактирования\n";
16.     std::cout << "Введите команду: ";
17.     std::cin >> choice;
18.     getchar();
19.     std::cout << std::endl;
20.     while(choice != '4')
21.     {
22.         switch(choice)
23.         {
24.             case '1':
25.                 std::cout << "Введите элемент: ";
26.                 std::cin >> element;
27.                 getchar();
28.                 Tree.searchAndInsertElement(element);
29.                 std::cout << std::endl;
30.                 break;
31.             case '2':
32.                 std::cout << "Введите элемент: ";
33.                 std::cin >> element;
34.                 getchar();
35.                 Tree.deleteElement(element);
36.                 std::cout << std::endl;
37.                 break;
38.             case '3':
39.                 Tree.printTree(0);
40.                 std::cout << std::endl;
41.                 break;
42.             case '4':
43.                 std::cout << "Выход";
44.                 break;
45.             default:
46.                 std::cout << "ОШИБКА: Неизвестная команда. Введите корректный
    номер команды." << std::endl;
47.                 break;
48.         }
49.         if (choice != '4')
50.         {
51.             std::cout << "1 - Добавить, 2 - Удалить, 3 - Печать, 4 - Выход" <<
    std::endl;
52.             std::cout << "Введите команду: ";
53.             std::cin >> choice;
54.             getchar();
55.             std::cout << std::endl;
56.         }
57.     }
58.     std::cout << "Конечное дерево:" << std::endl;
59.     Tree.printTree(0);
```

```
60.  }
61.
62.  int readFromFile(const std::string& str, RandomBinarySearchTree<char> Tree)
63.  {
64.      std::string str1;
65.      char element;
66.      std::cout << "Для файла: " << str << std::endl;
67.      std::ifstream inputFile(str);
68.      if (!inputFile.is_open())
69.      {
70.          std::cout << "ОШИБКА: Файл не открыт." << std::endl;
71.          return 0;
72.      }
73.      if (inputFile.eof())
74.      {
75.          std::cout << "ОШИБКА: Файл пуст." << std::endl;
76.          return 0;
77.      }
78.      while (inputFile >> element)
79.          Tree.searchAndInsertElement(element);
80.      Tree.printTree(0);
81.      readFromConsole(Tree);
82.      inputFile.close();
83.      return 0;
84.  }
85.
86.  int main(int argc, char* argv[])
87.  {
88.      setlocale(LC_ALL, "Russian");
89.      RandomBinarySearchTree<char> Tree;
90.      std::cout<< "_____________________________________________\n"
91.                  "         Курсовая работа на тему:\n"
92.                  "            РАНДОМИЗИРОВАННОЕ\n"
93.                  "                БИНАРНОЕ\n"
94.                  "                 ДЕРЕВО\n"
95.                  "                 ПОИСКА\n"
96.                  "_____________________________________________\n"
97.                  "                     студент: Артем Бутко\n"
98.                  "                      группа:        8304\n"
99.                  "_____________________________________________\n"<< std::endl;
100.
101.     if (argc == 1)
102.     {
103.         char choice = 0;
104.         std::string str;
105.         std::cout << "Откуда Вы хотите считать бинарное дерево?" << std::endl;
106.         std::cout << "    1) Из файла.\n"
107.                      "    2) С консоли (пользовательский ввод).\n"
108.                      "    3) Выйти из программы.\n";
109.         std::cout << "Введите команду: ";
110.         std::cin >> choice;
111.         getchar();
112.         std::cout << std::endl;
113.         while (choice != '3')
114.         {
115.             switch (choice)
116.             {
117.                 case '1':
118.                     std::cout << "___________Считывание из файла__________" <<
    std::endl;
119.                     std::cout << "Введите путь до файла: ";
120.                     std::getline(std::cin, str);
121.                     readFromFile(str, Tree);
122.                     choice = '3';
123.                     break;
```

```
124.                 case '2':
125.                     std::cout << "___________Считывание с консоли__________" <<
    std::endl;
126.                     readFromConsole(Tree);
127.                     choice = '3';
128.                     break;
129.                 case '3':
130.                     std::cout << "Выход из программы." << std::endl;
131.                     break;
132.                 default:
133.                     std::cout << "ОШИБКА: ОШИБКА: Неизвестная команда. Введите
    корректный номер команды." << std::endl;
134.                     break;
135.             }
136.             if (choice != '3')
137.             {
138.                 std::cout << "Введите команду: ";
139.                 std::cin >> choice;
140.                 getchar();
141.                 std::cout << std::endl;
142.             }
143.         }
144.
145.     }
146.     else readFromFile(argv[1], Tree);
147.     return 0;
148. }
```

ПРИЛОЖЕНИЕ Б

ФАЙЛ RBST.H

```
1.  #pragma once
2.  #include <random>
3.  #include <cmath>
4.
5.  template <class Elem>
6.  class RandomBinarySearchTree
7.  {
8.  private:
9.      struct Node
10.     {
11.         Elem element;
12.         std::shared_ptr<Node> left;
13.         std::shared_ptr<Node> right;
14.         int N;
15.         explicit Node (Elem value)
16.         {
17.             element = value,
18.             left = right = 0;
19.             N = 1;
20.         }
21.     };
22.     typedef std::shared_ptr<Node> link;
23.     link head;
24.
25.     link search(link root, Elem value)
26.     {
27.         if (!root)
28.         {
29.             std::cout << " * искомый ключ " << value << " не существует в дереве.
    *" << std::endl;
30.             return 0;
31.         }
32.         if (value == root->element)
33.         {
34.             std::cout << " * искомый ключ " << value << " найден в дереве. *" <<
    std::endl;
35.             return root;
36.         }
37.         if (value < root->element)
38.         {
39.             std::cout << " * искомый ключ " << value << " меньше, чем ключ
    текущего узла " <<
40.             root->element << ", спускаемся на левое поддерево. *"<< std::endl;
41.             return search(root->left, value);
42.         }
43.         else
44.         {
45.             std::cout << " * искомый ключ " << value << " больше, чем ключ
    текущего узла " <<
46.             root->element << ", спускаемся на правое поддерево. *"<< std::endl;
47.             return search(root->right, value);
48.         }
49.     }
50.
51.     int getSize(link root)
52.     {
53.         if (!root) return 0;
54.         return root->N;
55.     }
56.
57.     void fixN(link root)
```

```
58.     {
59.         root->N = getSize(root->left) + getSize(root->right) + 1;
60.         std::cout << " * фиксируем размер дерева равный " << root->N << ". *" <<
    std::endl;
61.     }
62.
63.     link rotateRight(link root)
64.     {
65.         link tmp = root->left;
66.         if (!tmp) return root;
67.         root->left = tmp->right;
68.         tmp->right = root;
69.         tmp->N = root->N;
70.         fixN(root);
71.         std::cout << " * производим правую ротацию *" << std::endl;
72.         std::cout << " ** дерево до ротации: **" << std::endl;
73.         print(root, 0);
74.         std::cout << " ** дерево после ротации: **" << std::endl;
75.         print(tmp, 0);
76.         return tmp;
77.     }
78.
79.     link rotateLeft(link root)
80.     {
81.         link tmp = root->right;
82.         if (!tmp) return root;
83.         root->right = tmp->left;
84.         tmp->left = root;
85.         tmp->N = root->N;
86.         fixN(root);
87.         std::cout << " * производим левую ротацию *" << std::endl;
88.         std::cout << " ** дерево до ротации: **" << std::endl;
89.         print(root, 0);
90.         std::cout << " ** дерево после ротации: **" << std::endl;
91.         print(tmp, 0);
92.         return tmp;
93.     }
94.
95.     link insertRoot(link root, Elem value)
96.     {
97.         if (!root)
98.         {
99.             std::cout << "Узел " << value << " подвешен." << std::endl;
100.            return std::unique_ptr<Node>(new Node(value));
101.        }
102.        if (root->element > value)
103.        {
104.            std::cout << " * добавляемый ключ " << value << " меньше, чем ключ
    текущего узла " <<
105.            root->element << ", спускаемся на левое поддерево и производим правую
    ротацию. *"<< std::endl;
106.            root->left = insertRoot(root->left, value);
107.            return rotateRight(root);
108.        }
109.        else
110.        {
111.            std::cout << " * добавляемый ключ " << value << " больше, чем ключ
    текущего узла " <<
112.            root->element << ", спускаемся на правое поддерево и производим левую
    ротацию. *"<< std::endl;
113.            root->right = insertRoot(root->right, value);
114.            return rotateLeft(root);
115.        }
116.    }
117.
```

```
118.     link insert(link root, Elem value)
119.     {
120.         if (!root)
121.         {
122.             std::cout << "Узел " << value << " подвешен." << std::endl;
123.             return std::unique_ptr<Node>(new Node(value));
124.         }
125.         if (rand()%(root->N+1) == 0)
126.         {
127.             std::cout << " * добавляемый ключ " << value << " вставляется на место
    текущего корня " <<
128.                       root->element << " с вероятностью "<< 1.0 / (1.0 +
    (double)root->N) << ". *" << std::endl;
129.             return insertRoot(root, value);
130.         }
131.         if (root->element > value)
132.         {
133.             std::cout << " * добавляемый ключ " << value << " меньше, чем ключ
    текущего узла " <<
134.             root->element << ", спускаемся на левое поддерево с вероятностью " <<
    (double)root->N / (1.0 + (double)root->N) << ". *" << std::endl;
135.             root->left = insert(root->left, value);
136.         }
137.         else
138.         {
139.             std::cout << " * добавляемый ключ " << value << " больше, чем ключ
    текущего узла " <<
140.             root->element << ", спускаемся на правое поддерево с вероятностью " <<
    (double)root->N / (1.0 + (double)root->N) << ". *"<< std::endl;
141.             root->right = insert(root->right, value);
142.         }
143.         fixN(root);
144.         return root;
145.     }
146.
147.     link merge(link left, link right)
148.     {
149.         if(!left) return right;
150.         if(!right) return left;
151.         if(rand()%(left->N + right->N) < left->N)
152.         {
153.             std::cout << " * корень " << left->element << " (левый) был выбран
    корнем нового дерева с вероятностью "
154.             << (double) left->N / (double)(left->N + right->N) << ". *" <<
    std::endl;
155.             left->right = merge(left->right, right);
156.             fixN(left);
157.             return left;
158.         }
159.         else
160.         {
161.             std::cout << " * корень " << right->element << " (правый) был выбран
    корнем нового дерева с вероятностью "
162.             << (double) right->N / (double)(left->N + right->N) << ". *" <<
    std::endl;
163.             right->left = merge(left, right->left);
164.             fixN(right);
165.             return right;
166.         }
167.     }
168.
169.     link remove(link root, Elem value)
170.     {
171.         if(!root)
172.         {
```

```
173.             std::cout << " * удаляемый ключ " << value << " не существует в
    дереве. *" << std::endl;
174.             return root;
175.         }
176.         if(root->element == value)
177.         {
178.             std::cout << " * удаляемый ключ " << value << " найден в дереве,
    начинается процедура удаления. *" << std::endl;
179.             link tmp = merge(root->left, root->right);
180.             return tmp;
181.         }
182.         else if(value < root->element)
183.         {
184.             std::cout << " * удаляемый  ключ " << value << " меньше, чем ключ
    текущего узла " <<
185.                         root->element << ", спускаемся на левое поддерево. *"<<
    std::endl;
186.             root->left = remove(root->left, value);
187.         }
188.         else
189.         {
190.             std::cout << " * удаляемый  ключ " << value << " больше, чем ключ
    текущего узла " <<
191.                         root->element << ", спускаемся на правое поддерево. *"<<
    std::endl;
192.             root->right = remove(root->right, value);
193.         }
194.         return root;
195.     }
196.
197.     void print(link root, int i)
198.     {
199.         if (root->right != 0) print(root->right, i+1);
200.         for (int j = 0 ; j < i; ++j)
201.             std::cout << " - ";
202.         std::cout << "(" << root->element << ")" << std::endl;
203.         if (root->left != 0) print(root->left, i+1);
204.     }
205.
206. public:
207.     RandomBinarySearchTree()
208.     {
209.         head = 0;
210.     }
211.
212.     void searchAndInsertElement(Elem value)
213.     {
214.         std::cout << " * начинается поиск элемента " << value << " среди элементов
    РБДП *" << std::endl;
215.         if (!search(head, value))
216.         {
217.             std::cout << "Ключ " << value << " не был найден в дереве." <<
    std::endl;
218.             std::cout << " * начинается вставка элемента " << value << " в РБДП *"
    << std::endl;
219.             head = insert(head, value);
220.         }
221.         else std::cout << "Ключ " << value << " уже существует в дереве." <<
    std::endl;
222.     }
223.
224.     void deleteElement(Elem value)
225.     {
226.         head = remove(head, value);
227.     }
```

```
228.
229.     void printTree(int i)
230.     {
231.         if(!head) std::cout << "Дерево не существует." << std::endl;
232.         else print(head, i);
233.     }
234.
235. };
```

ПРИЛОЖЕНИЕ В

ТЕСТИРОВАНИЕ. СЧИТЫВАНИЕ ИЗ ФАЙЛА.

```
____________Считывание из файла__________
Введите путь до файла: /Users/artembutko/Desktop/programming/CLionProjects/RBTS/cmake-build-debug/test.txt
Для файла: /Users/artembutko/Desktop/programming/CLionProjects/RBTS/cmake-build-debug/test.txt
 * начинается поиск элемента 1 среди элементов РБДП *
 * искомый ключ 1 не существует в дереве. *
Ключ 1 не был найден в дереве.
 * начинается вставка элемента 1 в РБДП *
Узел 1 подвешен.
 * начинается поиск элемента 2 среди элементов РБДП *
 * искомый ключ 2 больше, чем ключ текущего узла 1, спускаемся на правое поддерево. *
 * искомый ключ 2 не существует в дереве. *
Ключ 2 не был найден в дереве.
 * начинается вставка элемента 2 в РБДП *
 * добавляемый ключ 2 больше, чем ключ текущего узла 1, спускаемся на правое поддерево с вероятностью 0.5. *
Узел 2 подвешен.
 * фиксируем размер дерева равный 2. *
 * начинается поиск элемента 3 среди элементов РБДП *
 * искомый ключ 3 больше, чем ключ текущего узла 1, спускаемся на правое поддерево. *
 * искомый ключ 3 больше, чем ключ текущего узла 2, спускаемся на правое поддерево. *
 * искомый ключ 3 не существует в дереве. *
Ключ 3 не был найден в дереве.
 * начинается вставка элемента 3 в РБДП *
 * добавляемый ключ 3 больше, чем ключ текущего узла 1, спускаемся на правое поддерево с вероятностью 0.666667. *
 * добавляемый ключ 3 больше, чем ключ текущего узла 2, спускаемся на правое поддерево с вероятностью 0.5. *
Узел 3 подвешен.
 * фиксируем размер дерева равный 2. *
 * фиксируем размер дерева равный 3. *
 * начинается поиск элемента 4 среди элементов РБДП *
 * искомый ключ 4 больше, чем ключ текущего узла 1, спускаемся на правое поддерево. *
 * искомый ключ 4 больше, чем ключ текущего узла 2, спускаемся на правое поддерево. *
 * искомый ключ 4 больше, чем ключ текущего узла 3, спускаемся на правое поддерево. *
 * искомый ключ 4 не существует в дереве. *
Ключ 4 не был найден в дереве.
 * начинается вставка элемента 4 в РБДП *
 * добавляемый ключ 4 больше, чем ключ текущего узла 1, спускаемся на правое поддерево с вероятностью 0.75. *
 * добавляемый ключ 4 больше, чем ключ текущего узла 2, спускаемся на правое поддерево с вероятностью 0.666667. *
 * добавляемый ключ 4 вставляется на место текущего корня 3 с вероятностью 0.5. *
 * добавляемый ключ 4 больше, чем ключ текущего узла 3, спускаемся на правое поддерево и производим левую ротацию. *
Узел 4 подвешен.
 * фиксируем размер дерева равный 1. *
 * производим левую ротацию *
 ** дерево до ротации: **
(3)
 ** дерево после ротации: **
(4)
 - (3)
 * фиксируем размер дерева равный 2. *
 * фиксируем размер дерева равный 3. *
 * начинается поиск элемента 2 среди элементов РБДП *
 * искомый ключ 2 больше, чем ключ текущего узла 1, спускаемся на правое поддерево. *
 * искомый ключ 2 найден в дереве. *
Ключ 2 уже существует в дереве.
 * начинается поиск элемента 1 среди элементов РБДП *
 * искомый ключ 1 найден в дереве. *
Ключ 1 уже существует в дереве.
 -  - (4)
 -  -  - (3)
 - (2)
(1)
```

Рисунок В.1 — Результат считывания РБДП из файла